

# Domain Based Storage Protection with Secure Access Control for the Cloud

Nicolae Paladi  
Swedish Institute of Computer  
Science  
Stockholm, Sweden  
[nicolae@sics.se](mailto:nicolae@sics.se)

Antonis Michalas  
Swedish Institute of Computer  
Science  
Stockholm, Sweden  
[antonis@sics.se](mailto:antonis@sics.se)

Christian Gehrman  
Swedish Institute of Computer  
Science  
Stockholm, Sweden  
[chrisg@sics.se](mailto:chrisg@sics.se)

## ABSTRACT

Cloud computing has evolved from a promising concept to one of the fastest growing segments of the IT industry. However, many businesses and individuals continue to view cloud computing as a technology that risks exposing their data to unauthorized users. We introduce a data confidentiality and integrity protection mechanism for Infrastructure-as-a-Service (IaaS) clouds, which relies on trusted computing principles to provide transparent storage isolation between IaaS clients. We also address the absence of reliable data sharing mechanisms, by providing an XML-based language framework which enables clients of IaaS clouds to securely share data and clearly define access rights granted to peers. The proposed improvements have been prototyped as a code extension for a popular cloud platform.

## Categories and Subject Descriptors

K.6.5 [MANAGEMENT OF COMPUTING AND INFORMATION SYSTEMS]: Security and Protection—*Authentication*

## Keywords

Cloud Computing; Security; IaaS; Storage Protection

## 1. INTRODUCTION

Cloud computing continues its path towards wider adoption, and more companies attempt to tap into the promise of cost savings. Evidence to the success of the Infrastructure-as-a-Service (IaaS) model are both the increasing competition among IaaS cloud providers and the rush to migrate to IaaS clouds among businesses.

Moving traditional infrastructure to shared virtualized environments raises new security challenges. We can hope that users are aware of such security issues and strive to obtain from IaaS clouds security properties – such as execution isolation and control over data – which are on a par with on-site deployments. However, considering that clients of IaaS

clouds share execution and storage resources with other tenants, anonymous to them, currently available security solutions have proved to be insufficient. In [1], the authors have achieved to map the cloud infrastructure, collocate a malicious virtual machine (VM) instance with a target instance and launch side-channel attacks to extract information. The authors of [2] describe a range of attacks on management interfaces of public clouds using signature wrapping and XSS attacks. As a result, the attackers would be able to compromise the control interfaces of the IaaS cloud and misuse the cloud resources of other tenants. Finally, a recent example are the “dirty disks” of a public IaaS provider [3], where clients were able to read from improperly sanitised storage devices data stored by previous clients. This directly points to one of the unsolved problems in public IaaS clouds – ensuring data protection and secure data sharing.

Full-disk encryption has emerged as a solid solution for data confidentiality protection and is also mentioned in [3] as a solution to the “dirty disks” problem. However, full-disk encryption creates hurdles for data sharing, widely recognized as an essential feature for cloud applications [4]. Despite the variety of available open source cloud management platforms (e.g OpenStack, Eucalyptus, OpenNebula), allocation of read-write permissions for shared data between collaborating tenants still remains an open problem. In this paper we address the outlined gap. We improve and extend previous work by adding capabilities to both grant access to data to other IaaS cloud clients and assign access permissions.

### 1.1 Our Contribution

The contribution of this work is twofold. We first present a secure storage protection protocol that provides per-VM instance access control and allows the client to control a VM instance’s read and write access rights over a storage device at launch time. We introduce an XML-based language framework that allows users to define role-based access control in order to grant access, based on permissions, to other users in the IaaS cloud. Our protocol allows a granular access rights management per VM instance and storage device. In addition, we analyse our protocol and show it is resistant under malicious behaviours. Second, we complement the analysis with extensive experimental results that show the effectiveness of the protocol.

### 1.2 Organization

In Section 2, we review some of the most important protocols that provide domain storage protection in public IaaS

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

SCC’14, June 3, 2014, Kyoto, Japan.

Copyright 2014 ACM 978-1-4503-2805-0/14/06 ...\$15.00.

<http://dx.doi.org/10.1145/2600075.2600082>.

clouds and mechanisms for secure data sharing in clouds. In Section 3, we describe the problem of data protection in IaaS clouds and define the important terms used throughout the paper. In Section 4, we describe the system model of a cloud platform (*CP*) which stands at the basis of our protocol implementation. In Section 5, we present our protocol for secure storage protection data sharing mechanism in IaaS clouds. Section 7 contains experimental results of the protocol benchmarks, while Section 8 concludes the paper.

## 2. RELATED WORK

The importance of data confidentiality protection and isolation of data between IaaS cloud tenants is underlined by the attention it has received from the research community.

In [5], authors propose a full disk background encryption model by introducing TCVisor, a hypervisor with a paraspassthrough architecture that introduces *TPM* support and novel key-management approach. Support for *TPM* is added in order to store parts of cryptographic keys and whole-disk checksums for integrity checking. In addition to that, Merkle trees are used for integrity verification and protection of the root value relying on *TPM* functionality. However, the poor description of storing/sealing the root value of the Merkle tree hash, raises doubts about protocol's validity.

The authors of [6] focus on hypervisor-level data protection and introduce Cloudvisor – a security monitor underneath the commodity hypervisor which provides protection to the hosted VMs. CloudVisor runs in host mode and encrypts the data exchange between a VM and the hypervisor and verifies the integrity, freshness and ordering of disk I/O data. One immediate limitation of the solution in [6] are the severe functionality limitations, such as support for a single VM instance. Our protocol uses the functionality offered by commodity hypervisors in order to ensure data protection and does not introduce such severe limitations.

A solution for management of encrypted data is described in [7], where each information block is encrypted with a different symmetric key, thus aiming for a cryptography-based access control. An ‘information block’ represents an abstract concept of arbitrary size. The paper assumes a lazy revocation model, where a user indefinitely maintains access to the data that she could reach prior to revocation (regardless of whether or not the data has been accessed before access revocation). While similar to our model in aspects such as information blocks and encryption with different symmetric keys, we propose an active revocation model, where the keys can not be retrieved once the access is revoked.

Few of the IaaS storage protection schemes address the problem of sharing files with certain permissions. In [8], authors analysed access rights management of shared versioned encrypted data on cloud infrastructure for a restricted group. In their model they proposed an adoption for enabling scalable and flexible key management within cloud. By representing access rights as a graph and based on [9], authors were able to distinguish between the keys used for encrypting data and the encrypted updates on the keys. Thus, enabling flexible join/leave operations of clients. Despite being an attractive approach, the requirement for client-side encryption limits the applicability of the scheme and ignores the limitations to functionality (such as indexing and search) that it introduces. In our model all cryptographic operations

are performed on trusted IaaS compute hosts, which are able to allocate more computational resources than client devices.

Data-Protection-as-a-Service (DPaaS) [4] is a conceptual architecture which aims to address the need for integrity, privacy, access transparency, ease of verification and rich computation in a cloud environment. DPaaS recognises the difficulties with full disk encryption and focuses on data sharing, proposing flexible data units access control lists. Despite highlighting a range of important issues related to cloud data protection, DPaaS falls short of proposing a clear implementation strategy and specific sharing mechanisms that could be used by cloud tenants. In the current paper, we address many of the concerns highlighted in [4], propose an XML-based framework to enable data sharing and describe a test implementation in the context of a cloud platform.

## 3. PRELIMINARIES

Our protocol assumes that basic functionality normally provided by a *CP*, such as registration and authentication of a user, is available. Similar to [10], the active parties in our protocol are domain managers (*d*), virtual machines (*VM*), a secure component (*SC*) as well as a trusted third party (*TTP*). Domain managers can launch new VM instances, which can in turn create data and securely share it with other VM instances both within the same and other IaaS clouds. The proposed protocol also relies on the principles of trusted computing and capabilities of the Trusted Platform Module (*TPM*) [11].

For the purposes of our protocol, each domain manager, *SC* and *TTP* has a public/private key pair (*pk/sk*). The private key is kept secret, while the public key is shared with the community. Furthermore, we assume that during the initialization phase, each entity obtains a certificate via the certification authority provided by the *CP*. These keys and certificates will be used to protect internal message exchanges and hence the communication between the parties assumed to be secure. Finally, our protocol also relies on pseudorandom functions [12] – a major tool for the design of shared key cryptography protocols – to create symmetric keys.

Next, we define the main components of our protocol.

### *Disk encryption subsystem.*

The disk encryption subsystem is a software or hardware component responsible for encryption and decryption of data during respectively writes or reads from a storage device. It can encrypt storage units such as whole hard drives, partitions, software RAID volumes, logical volumes, and files. For simplicity, this paper assumes a software-based disk encryption subsystem, such as *dm-crypt*, a popular open-source disk encryption subsystem which uses the Linux kernel Crypto API.

### *Domain Manager ( $d_i$ ).*

Domain Managers are responsible for launching virtual machines and handling the VM instances that they create. Let  $DM = \{d_1, \dots, d_n\}$  be the set of all domain managers in our IaaS cloud. Then, the set of all VMs that each domain manager  $d_i$  owns is defined as  $VM_i = \{vm_1^i, \dots, vm_n^i\}$ .

### *Domain ( $Dom_i$ ).*

A domain is an abstract concept referring to a collection of data. A domain  $Dom_i$  can be created only from a domain manager which is also responsible for granting permissions to VM instances within the cloud environment. As a storage unit, a domain can be any unit supported by the disk encryption subsystem. Let  $D_i = \{Dom_1^i, \dots, Dom_n^i\}$  be the set of all domains created by a domain manager  $d_i$ .

### Trusted Platform Module (TPM).

*TPM* is a tamper-evident hardware cryptographic coprocessor which follows specifications of the Trusted Computing Group (*TCG*) [11]. In this work, we assume that the IaaS compute hosts are equipped with a *TPM* v1.2 chip. An active *TPM* records the software state of the platform at boot time and stores it in its platform configuration registers (PCRs) as a list of hashes. *TPM* enables data protection by securely maintaining cryptographic keys, as well as though the set of functions it exposes. The *bind* and *seal* functions are particularly relevant for the proposed solution. According to [11], a message encrypted (“bound”) using a particular *TPM*’s public key is decryptable only by using the private key of the same *TPM*. Sealing is a special case of the binding functionality, where the encrypted messages produced through binding are only decryptable in a certain platform state (defined by the PCR values) to which the message is *sealed*. This ensures that an encrypted message can only be decrypted by a platform found in a certain prescribed state. We refer to [11] for a detailed description of the bind and seal operations.

### Trusted Third Party (TTP).

In this paper we assume a “trusted third party”, which is trusted by the community and plays a key role in our protocol. We rely on the commonly supported proposition that a large code base normally contains a proportionally large number of vulnerabilities [13]. To reduce the code base, it is important that the *TTP* only supports the minimal necessary functionality. *TTP* is able to communicate with components deployed on compute hosts to exchange integrity attestation information, authentication tokens and cryptographic keys. In addition, *TTP* can attest platform integrity based on the integrity attestation quotes provided by the *TPM* on the respective compute hosts, as well as seal data to a trusted configuration of the hosts. Finally, *TTP* can verify the authenticity of a client as well as perform necessary cryptographic operations.

### Secure Component (SC).

*SC* is a verifiable execution module which performs confidentiality and integrity protection operations on guest VM instance data. *SC* is present on all compute hosts and acts as a mediator between the *CP* and the *TTP*. *SC* is responsible for forwarding the requests of domain managers to either the *TTP* or the disk encryption subsystem, depending on the type of request. In addition, *SC* is the only entity from which *TTP* accepts requests.

**DEFINITION 1 (PSEUDORANDOM FUNCTION).** Let *PRF*  $(K, c)$  be a family of functions<sup>1</sup> with two inputs, a secret key

<sup>1</sup>A function family is a map  $F : \mathcal{K} \times \mathcal{D} \rightarrow \mathcal{R}$ , where  $\mathcal{K}$  is the set of keys of  $F$ ,  $\mathcal{D}$  is the domain of  $F$  and  $\mathcal{R}$  is the range of  $F$ . The two-input function  $F$  takes a key  $K$  and an input  $X$  to return a point  $Y$  denoted by  $F(K, X)$ .

$K$  and a content  $c$ . We say that *PRF* is a pseudorandom function iff the input-output behavior of a random instance of the family is “computationally indistinguishable” from that of a random function.

In this paper, we focus on the following problem:

**Problem Statement:** A domain manager  $d_i$ , operates a set of VM instances  $VM_i = \{vm_1^i, \dots, vm_n^i\}$ . In addition to that,  $d_i$  operates a set of domains  $D_i = \{Dom_1^i, \dots, Dom_k^i\}$  made available to the VM instances as storage devices. Finally, a different domain manager  $d_j$  operates a set of VMs  $VM_j = \{vm_1^j, \dots, vm_n^j\}$ . We aim to create secure mechanisms that will satisfy the following requirements:

- Data stored in each  $Dom_i^i$  should be encrypted;
- Plaintext data from each  $Dom_i^i$  should be revealed only to VM instances with corresponding access privileges;
- Access privileges for members of  $VM_i$  to domains in  $D_i$  should be exclusively controlled by domain manager  $d_i$ ;
- $d_i$  should be able to share access privileges for domains in  $D_i$  to other domain managers, e.g.  $d_j$ ;

### Adversarial Model.

Similar to existing works in the area, we assume that the adversary is acting under the Dolev-Yao adversarial model [14]. In this model, malicious nodes can overhear all messages and may attempt to use them to learn information that should otherwise remain private. Adversaries can also create, replay and destroy messages; however, they are not able to break any cryptographic mechanism.

The notation  $E_i(\cdot)$  will refer to the results of the application of an asymmetric encryption function that only entity  $i$  can decrypt with her private key.

## 4. IAAS CLOUD SYSTEM MODEL

We consider an IaaS cloud model as defined by the NIST, where an IaaS cloud provides “processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications.” [15]. The model is based on a *CP* deployed on multiple server platforms. The *CP* is a distributed middleware composed of a series of *management services* on management hosts and corresponding *service agents* deployed on service hosts.

Service hosts can be dedicated to compute resources (*compute hosts* hosting virtual machines) and/or storage resources. Management services control vital aspects of the *CP* such as scheduling, networking, identity, volume and virtual machine image management. In a typical *CP* architecture, management services and service clients communicate among themselves using the advanced message queuing protocol (AMQP) based on a publish-subscribe model. The capabilities of a *CP* are exposed to *domain managers* through a set of APIs, graphical or command line interfaces. Domain managers use the functionality of the *CP* in order to operate VM instances, create storage volumes and custom network topologies using software defined networks.

The IaaS cloud is maintained by a *cloud service provider*, an organization responsible for the operation of the IaaS

cloud. The cloud service provider can be either private or public. In this paper we assume a public cloud provider, with multiple domain managers sharing physical resources through a virtualization layer. On the physical compute host level, virtualization between the domain managers is ensured by the hypervisor; communication isolation is ensured through VLAN tagging (using the IEEE 801.2Q tags); *CP* level isolation relies on the authentication service, which authenticates domain managers based on their credentials.

Domain managers can create and attach block storage volumes to one or more virtual machine instances in the cloud environment. Support for storage encryption is offered by a standard disk encryption subsystem. Domain managers can also grant access rights on a certain volume to their peers.

## 5. PROTOCOL DESCRIPTION

Our work is an extension of the protocol presented in [10] where the authors introduced the principles of “domain-based storage protection” (DBSP) in a public IaaS cloud. DBSP is based on a set of protocols that allow an IaaS client to shift the responsibility for data confidentiality and integrity to an external *TTP* – away from the IaaS provider. This approach relies on two protocols: initial data write operation and subsequent data read and write operations. The core idea of this approach is to store information necessary to derive the decryption key for a given data volume in a header appended to the volume itself. The decryption key can only be derived by the *TTP* using the information stored at write time in the volume header and *TTP*’s own secret key. Besides withdrawing data protection responsibility from the IaaS provider, this enables a fluid migration of the IaaS client’s encrypted data assets to a different IaaS provider, while maintaining trust in the same *TTP*. In addition, the approach in [10] allows to precede the release of the decryption key with a remote attestation of the platform done by the same *TTP*. Remote attestation would: (i) ensure that the execution platform is in a certain trusted state and (ii) allow the *TTP* to seal the decryption key to the trusted configuration of the host to prevent its misuse in the form of migration to other platforms or usage in a different platform configuration.

Having briefly covered the background, we proceed with a high-level overview of our protocol. A domain manager  $d_i$  launches  $n$  VM instances and a set of domains  $D_i = \{Dom_1^i, \dots, Dom_n^i\}$  that the VM instances can access to read and write data. To this end,  $d_i$  authenticates to the *CP* and requests to generate a domain  $Dom_k^i$ . The request also describes the VM instances belonging to the set  $VM_i$  that should have access to the specified domain and the respective access rights. The *CP* is responsible for creating  $Dom_k^i$  and allocating the corresponding disk space. During this process, the *SC* (part of the *CP*) contacts *TTP*, to generate a symmetric key ( $K_{Dom_k^i}^2$ ) that will be used to encrypt data in  $Dom_k^i$ . Following the successful creation of  $Dom_k^i$ , a domain manager must prove the right of a certain VM instance to access  $Dom_k^i$ .

In the following protocol, the participants exchange a number of messages. In order to ensure the integrity of the com-

munication, we assume that each message is signed by the sender and the receiver can easily verify it.

### 5.1 Domain Sharing

One of the challenges of cloud computing is to enable users to securely administer data in a shared environment. Despite the fact that the protocol in [10] achieves protection of data in the cloud, it is considered a rudimentary work since it lacks sharing functionality. In the following paragraphs, we bridge this gap by presenting an extension of the protocol introduced in [10], which can be added to a typical *CP* to allow a domain manager to share a storage domain with other VM instances in the IaaS cloud.

#### Domain Registration.

Assume that a domain manager  $d_i$  wishes to create a domain  $Dom_k^i$ . As a first step,  $d_i$  defines the parameters needed to create the domain (e.g volume, size, name, etc.) and a description of the type of data stored in  $Dom_k^i$ . This description constitutes the metadata ( $meta_k^i$ ) of  $Dom_k^i$  and will be used for domain discovery and data search. Upon receiving a domain creation request, the *CP* generates an XML document (Figure 1), allocates the corresponding disk space (by e.g. creating a logical volume) and adds  $meta_k^i$  to the header of the allocated volume. It also adds the domain credential that will later be used by the *TTP* to the header of the allocated volume.

```

1 <DomainCredential scope="create">
2   <CredentialID>cred:id</CredentialID>
3   <Timestamp>issue:time</Timestamp>
4 </DomainCredential>
5 <DomainDescription>
6   <DomainID Encoding="xmlenc:rsa">Dom_k^i</DomainID>
7   <DomainName lang="EN">dom:name</DomainName>
8   <DomainManager Encoding="xmlenc:rsa">
9     E_TTP(d_i)
10   </DomainManager>
11   <DomainVolume Encoding="xmlenc:rsa">
12     E_d_i(volume:id)
13   </DomainVolume>
14   <DomainSize>E_d_i(disk:size)</DomainSize>
15   <Metadata>meta_k^i</Metadata>
16 </DomainDescription>

```

**Figure 1: Credential specification for the creation of a new domain.**

Once a domain has been created,  $d_i$  can grant access permissions to multiple VM instances. We analyse the problem of sharing a domain in the following two use cases:

**A. Grant access to  $Dom_k^i$  for a VM instance in the set  $VM_i$ :** Assume  $d_i$  intends to grant access to  $Dom_k^i$  for a VM instance  $vm_i^j$ , which is part of the set  $VM_i$ . To do this,  $d_i$  requests the launch of a new virtual machine  $vm_i^j$  and defines the access domain(s) and respective permissions for  $vm_i^j$ . More precisely,  $d_i$  generates and sends to the *CP* an XML document as shown in Figure 2 where each encrypted element (*i.e.* *xmlenc:rsa*) is protected with  $pk_{TTP}$  and thus *TTP* is the only one who can decrypt it. Prior to launching the VM instance with the requested domain(s) attached, *SC* checks that element *DomainDescription* matches the one stored in the header of the domain  $Dom_k^i$ . If it does, *SC* updates the XML structure of Figure 1 by adding the element *VirtualMachine* contained in the VM instance launch request.

<sup>2</sup>All data in a single domain is protected with the same storage protection master key, the domain key. This key is generated by the *TTP* and cannot ever leave *TTP*’s logical perimeter.

```

1 <DomainCredential scope="addVM">
2   <CredentialID>cred:id</CredentialID>
3   <Timestamp>issue:time</Timestamp>
4   <DomainDescription>
5     <DomainID>Domki</DomainID>
6     <DomainName lang="EN">dom:name</DomainName>
7     <DomainManager>manager:id</DomainManager>
8   </DomainDescription>
9   <VirtualMachine manager="di">
10     <VMID Encoding="xmlenc:rsa">ETTP(vmli)</VMID>
11     <Nonce Encoding="xmlenc:rsa">ETTP(r)</Nonce>
12     <permissions Encoding="xmlenc:rsa">
13       <permission>r</permission>
14       <permission>w</permission>
15     </permissions>
16   </VirtualMachine>
17 </DomainCredential>

```

**Figure 2: Credential Specification for the addition of a VM to a domain.**

Once  $vm_l^i$  is launched,  $d_i$  generates a credential as shown in Figure 3 that will be used later to prove that  $d_i$  has granted access permissions for  $Dom_k^i$  to  $vm_l^i$ .

```

1 <DomainCredential scope="accessDomain">
2   <CredentialID>cred:id</CredentialID>
3   <Timestamp>issue:time</Timestamp>
4   <DomainDescription>
5     <DomainID Encoding="xmlenc:rsa">dom:id</DomainID>
6     <DomainName lang="EN">dom:name</DomainName>
7     <DomainManager>manager:id</DomainManager>
8     <Metadata Encoding="xmlenc:rsa">ETTP(metaki)</
9       Metadata>
10   </DomainDescription>
11   <VirtualMachine manager="di">
12     <VMID Encoding="xmlenc:rsa">ETTP(vmli)</VMID>
13     <Nonce Encoding="xmlenc:rsa">ETTP(r)</Nonce>
14   </VirtualMachine>
15 </DomainCredential>

```

**Figure 3: Credential for presentation of granted permissions for a domain.**

**B. Grant access to  $Dom_k^i$  for a VM instance in the set  $VM_j$ :** Assume  $d_i$  intends to grant access to  $Dom_k^i$  to a VM instance  $vm_l^j$ , which is part of the set  $VM_j$  (operated by domain manager  $d_j$ ). As we have mentioned earlier, a VM instance must receive a credential from the corresponding domain manager in order to access files in a specific domain. In this case though, the manager of  $Dom_k^i$  is not the owner of  $vm_l^j$ , so in order to grant access to  $vm_l^j$ ,  $d_j$  requests a valid credential from  $d_i$ . Upon reception – if  $d_i$  accepts to give access to  $vm_l^j$  – it generates a credential as described in Figure 2 and sends it to the  $CP$ . Domain manager  $d_i$  also generates a random nonce  $r_j$  and sends  $E_{d_j}(r_j)$  to  $d_j$  as well as  $E_{SC}(r_j)$  to the  $CP$ . Upon reception,  $d_j$  decrypts it with  $sk_{d_j}$ , and sends it to  $CP$  which validates that  $D(E_{SC}(r_j)) = D(E_{d_j}(r_j))$ . Then,  $SC$  adds the corresponding VM to the credential of  $Dom_k^i$  as described in the previous case. More exactly,  $SC$  will add a nonce, the  $VMID$  and the access permissions (presented in Figure 4) to the credential of  $Dom_k^i$  (XML document in Figure 1).

## 5.2 Domain Access

Next, we describe the domain confidentiality protection mechanism and present the protocol to retrieve encryption keys and provide access to plain text data for authorized VM instances.

We assume that  $vm_l^i$  requires access to the data in  $Dom_k^i$ . To grant access,  $SC$  must retrieve from  $TTP$  the symmetric key ( $K_k^i$ ) used to confidentiality protect data in  $Dom_k^i$ . The

```

1 <VirtualMachine manager="di">
2   <VMID Encoding="xmlenc:rsa">ETTP(vmli)</VMID>
3   <Nonce Encoding="xmlenc:rsa">ETTP(r')</Nonce>
4   <permissions Encoding="xmlenc:rsa">
5     <permission>r</permission>
6   </permissions>
7 </VirtualMachine>

```

**Figure 4: Credential Specification for the addition of a VM to a domain.**

domain manager operating  $vm_l^i$  sends a request to  $CP$  in order to mount  $Dom_k^i$  to the virtual machine instance; the call is forwarded to and processed by  $SC$ . In the request,  $d_i$  sends the previously generated credential (Figure 3) and proves that  $vm_l^i$  has access to  $Dom_k^i$ .  $SC$  extracts from the header of the domain  $Dom_k^i$  a data structure (Figure 1) that contains information about the domain and sends it to  $TTP$  along with the unique identifier of  $vm_l^i$ .

As a first case, we assume that no data has been stored in  $Dom_k^i$  yet, which implies that  $K_k^i$  has not been generated yet. When  $TTP$  receives the message from  $SC$ , it first decrypts  $E_{TTP}(vm_l^i)$  from the XML document presented in Figure 3 and locates the  $ID$  of the VM instance contained in the credential. Next,  $TTP$  checks if the corresponding block (*i.e.* where  $VMID$  element is equal with  $vm_l^i$ ) exists in the credential of the domain. If it does,  $TTP$  decrypts the metadata and checks that values match in both XML files. It then finds the permissions of  $vm_l^i$  for  $Dom_k^i$  by decrypting  $permissions$  from the domain credential.

Once  $TTP$  has validated that  $vm_l^i$  is authorized to access  $Dom_k^i$ , it performs a remote attestation of the compute host where  $vm_l^i$  will be launched (for simplicity, we assume that this is also the source of the key request). The remote attestation involves obtaining a quote of the compute host's  $TPM$  platform configuration registers to evaluate whether the platform can be trusted. We leave out the minutiae of remote attestation and evaluation of platform trust level and refer the reader to [16].

In the event of a positive result of the  $TPM$  remote attestation,  $TTP$  generates a symmetric key ( $K_k^i$ ) that encrypts data in the domain. To create the key,  $TTP$  generates a random nonce  $r_k$  and evaluates the following:

$$K_k^i = PRF(meta_k^i || r_k, K_{TTP}),$$

where  $meta_k^i || r_k$  is respectively the concatenation of metadata and the random generated nonce, and  $K_{TTP}$  is a master key that does not leave the security perimeter of  $TTP$ . After generating the symmetric key for  $Dom_k^i$ ,  $TTP$  seals it to the trusted configuration of the compute host (similar to the key sealing procedures already described in [10, 16]) and returns to  $SC$  the response shown in Figure 5.

Upon receiving the message,  $SC$  first decrypts  $E_{SC}(vm_l^i)$  and checks if the request was sent from the VM instance contained in the response. If it was,  $SC$  calls the local  $TPM$  to unseal the key which – if the compute host remained in the earlier trusted state – reveals  $K_k^i$ .  $SC$  then uses it as input to the disk encryption subsystem on the compute host where  $vm_l^i$  is running. The disk encryption subsystem seamlessly decrypts the mounted volume hosting  $Dom_k^i$ . Next, the volume containing  $Dom_k^i$  is mounted as a disk device on  $vm_l^i$  – with read-write or read-only rights, depending on the permissions granted by the domain owner.



```

1 <response>
2 <DomainKey Encoding="xmlesc:rsa">
3    $E_{dmcrypt}(K_k^i)$ 
4 </DomainKey>
5 <VMID> $E_{SC}(vm_i^i)$ </VMID>
6 <Metadata>meta $_k^i$ || $E_{TTP}(r_k)$ </Metadata>
7 <permissions>
8 <permission>r</permission>
9 <permission>w</permission>
10 </permissions>
11 </response>

```

**Figure 5: Response of  $TTP$  after the generation of the domain symmetric key.**

The case where  $K_k^i$  has already been generated is similar, with the only difference that to recalculate  $K_k^i$ ,  $TTP$  will have to decrypt  $E_{TTP}(r_k)$  contained in the updated metadata and use it as an input to the pseudorandom function.

### 5.3 Revocation

There are cases when credentials of a VM may need to be revoked if a VM instance misbehaved, lost access rights to a domain, or permissions have been changed. In this section we describe the mechanism to change or revoke the permissions of a VM instance for a specific domain.

Following our previous scenario, we assume that  $d_i$  wants to change the access rights of  $vm_i^i$  for the domain  $Dom_k^i$ . We analyse the following two scenarios for  $d_i$ :

**A. Prevent  $vm_i^i$  from accessing  $Dom_k^i$ :** Assume  $d_i$  wants to completely remove  $vm_i^i$  from the list of VMs that are authorized to access  $Dom_k^i$ . First,  $d_i$  generates the XML file shown in Figure 6 and sends it to  $CP$ , which forwards the request to the  $SC$  on one of the host platforms. Upon reception,  $SC$  extracts the credential for  $Dom_k^i$  from the header of the volume and sends it to  $TTP$  along with the XML received from  $d_i$ .  $TTP$  decrypts  $E_{TTP}(vm_i^i)$  and finds the  $ID$  of the VM that should remove its access rights. Then,  $TTP$  finds the corresponding block in the XML that contains all the VM instances that have access to  $Dom_k^i$  and removes it. Finally,  $TTP$  returns to  $SC$  an updated XML document which does not contain  $vm_i^i$  and  $SC$  updates the header of  $Dom_k^i$  with the fresh credential.

```

1 <VMCredential scope="Revoke">
2 <CredentialID>cred:id</CredentialID>
3 <Timestamp>issue:time</Timestamp>
4 <DomainDescription>
5 <DomainID Encoding="xmlesc:rsa">dom:id</DomainID>
6 <DomainName lang="EN">dom:name</DomainName>
7 <DomainManager>manager:id</DomainManager>
8 </DomainDescription>
9 <VirtualMachine manager="d_i">
10 <VMID Encoding="xmlesc:rsa"> $E_{TTP}(vm_i^i)$ </VMID>
11 </VirtualMachine>
12 </VMCredential>

```

**Figure 6: Request to revoke the credential of a VM.**

**B. Change permissions of  $vm_i^i$  on  $Dom_k^i$ :** In this case we assume that  $d_i$  intends to just change the permission for  $vm_i^i$  ‘read-write’ to ‘read’. The procedure that is followed is identical to the one in scenario A.  $d_i$  generates a new credential for  $vm_i^i$  (Figure 7) and sends it to  $TTP$  via  $CP$ . Additionally,  $SC$  sends to  $TTP$  the credential of the domain that is stored in the header of the volume.  $TTP$  follows the same steps in order to update the credential of  $Dom_k^i$ . Following the successful update of the domain credential,  $SC$  sends the fresh credential to  $d_i$  who can use it in the

future in order to prove that  $vm_i^i$  is authorized to access the corresponding domain under certain permissions.

In both cases A and B,  $d_i$  has the option to receive a from  $TTP$  a confirmation proving that the permissions for  $vm_i^i$  have indeed been withdrawn or modified. To do this, prior to the request  $d_i$  generates a random nonce  $r_{rev}$ , encrypts it with  $TTP$  and sends it along with the credential of  $vm_i^i$ . Upon reception  $TTP$  – apart from altering the permissions of the corresponding VM – decrypts  $E_{TTP}(r_{rev})$  and returns to  $d_i$   $H(r_{rev}||vm_i^i)^3$ . Given that by definition the  $TTP$  will not deviate from the protocol, the returned hash is an implicit confirmation of the fact that  $TTP$  has received the update request and has modified the credential accordingly.

```

1 <VMCredential scope="UpdatePermissions">
2 <CredentialID>cred:id</CredentialID>
3 <Timestamp>issue:time</Timestamp>
4 <DomainDescription>
5 <DomainID Encoding="xmlesc:rsa">dom:id</DomainID>
6 <DomainName lang="EN">dom:name</DomainName>
7 <DomainManager>manager:id</DomainManager>
8 </DomainDescription>
9 <VirtualMachine manager="d_i">
10 <VMID Encoding="xmlesc:rsa"> $E_{TTP}(vm_i^i)$ </VMID>
11 <Nonce Encoding="xmlesc:rsa"> $E_{TTP}(r')$ </Nonce>
12 <permissions Encoding="xmlesc:rsa">
13 <permission>r</permission>
14 </permissions>
15 </VirtualMachine>
16 </VMCredential>

```

**Figure 7: Request for altering permissions of  $vm_i^i$  on domain  $Dom_k^i$ .**

## 6. SECURITY ANALYSIS

In this section, we analyse the behaviour of our protocol in several attack scenarios. In all of the attack scenarios, we assume that the involved parties follow the Dolev-Yao adversarial model [14] and can overhear all messages and may attempt to use them in order to learn information that otherwise should remain private or gain access to domains that are not authorized to.

**Unauthorized access to a domain:** Assume that a malicious domain manager  $d_m$  attempts to gain unauthorized access to a domain  $Dom_k^i$  for a VM instance  $vm_i^m$ . To do so, the domain manager will have to prove that she owns a credential for accessing  $Dom_k^i$ . The domain manager self-generates a credential and presents it to the  $CP$  in order to gain access to  $Dom_k^i$  (as shown in Figure 3). This can be easily done since the encrypted information contained in a credential is mainly generated by using the public key of  $TTP$ , which is also responsible for validating the correctness of the credential. As described in Section 5,  $SC$  retrieves the corresponding metadata from the header of  $Dom_k^i$  and forwards both artefacts to  $TTP$ . Upon reception,  $TTP$  verifies the correctness of the credential received from  $d_m$ . To this end,  $TTP$  decrypts the information contained in both artefacts and finds out that the  $ID$  of  $vm_i^m$  is not in the list of the authorized VM instances for the domain  $Dom_k^i$ . Thus, this attack cannot be launched.

**Using a valid credential from another domain manager:** In such a scenario we assume that a malicious domain manager  $d_m$  attempts to gain unauthorized access to a domain  $Dom_k^i$  for  $vm_i^m$  by providing a valid credential that belongs

<sup>3</sup> $H(.)$  is a secure cryptographic hash function such as  $SHA3$

to another VM instance, i.e. effectively impersonating the righteous domain manager. We assume that  $d_m$  gets a valid credential for  $Dom_k^i$  that was created for  $vm_k^i$ . This can be done in two different ways: either  $d_m$  can intercept a message in which  $d_i$  sends the credential to  $SC$  in order to access  $Dom_k^i$ ; or – if we assume that  $d_i$  is also acting maliciously –  $d_i$  can cooperate with  $d_m$ , and reveal to  $d_m$  the credential created for  $vm_k^i$ . In both cases,  $d_m$  will be able to convince  $TTP$  about the validity of the credential. Thus,  $TTP$  will first attest the trusted configuration of the host where the virtual machine  $vm_l^m$  will reside, then will calculate the domain key and will send back to  $SC$  the metadata showed in Figure 5. Upon reception,  $SC$  first decrypts  $E_{SC}(vm_k^i)$  and checks whether the domain manager requested to grant access to the VM instance stated in the response of the  $TTP$ . In the above attack scenario,  $SC$  will drop the request since the  $VMID$  received from  $TTP$  does not correspond to the VM instance for which access to  $Dom_k^i$  was requested. We can conclude that such an attack would only be possible if malicious domain managers can change their identity.

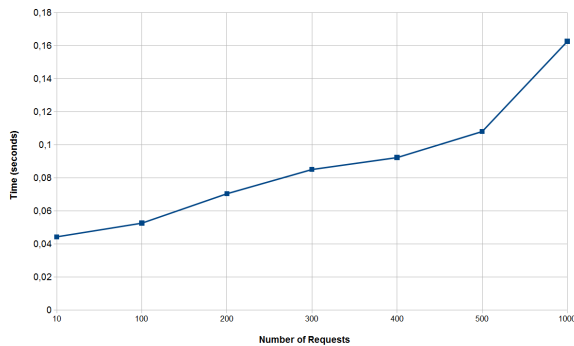
Using remote TPM attestation and the TPM seal operation, we obtain the confidence that  $SC$  will act according to the protocol and will verify that the requesting VM instance identifier matches the VM instance identifier authorized in the response obtained from the  $TTP$ . The domain decryption key is only made available to the target compute host with a trusted platform configuration, and can not be accessed in plain text once the compute host changes its platform configuration.

## 7. EXPERIMENTAL RESULTS

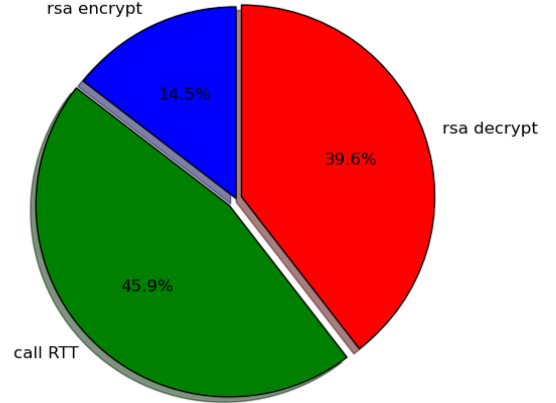
In order to measure the performance of the protocol, we have implemented the  $SC$  as a client and a server application serving the role of the  $TTP$ .

Our implementation follows the protocol:  $SC$  creates a request for an encryption key and sends it to  $TTP$ , which derives the encryption key and returns it in encrypted form together with the meta data.

The experiments aimed to analyse two main performance metrics: processing time and communication overhead. To this end, we ran several experiments, in order to measure the time to request a new encryption key, the duration of the most computation-intensive or network-intensive operations, as well as to measure the performance of  $TTP$ .



**Figure 8: Time required by  $TTP$  to process a request and to generate a key for a domain.**



**Figure 9: Proportion of execution time spent in functions during a key request.**

In the first phase of our experiments we measured the performance of  $TTP$  when serving multiple parallel requests from  $SC$ . We tested the time  $TTP$  needed in order to perform encryption/decryption operations, generate a domain key as well as to parse an XML for 10 to 1000 parallel requests. For encryption and decryption, we used the RSA cryptosystem with a key length of 1024 bits. Figure 8 illustrates the results in seconds as a function of the number of requests. As is evident from the graph, the required processing time is negligible and does not constitute any real burden to the functionality of the  $CP$ . We have found that, on average, the time needed for  $TTP$  to successfully respond to  $SC$  when receiving 1000 parallel requests is approximately equal to 0.16 seconds.

In the second phase of our experiments, we measured the communication delay for a single request sent to  $TTP$  by  $SC$  (with a sample of 1000 sequential requests), as well as the impact of domain key requests on the duration of the VM instance launch. Table 7 and Figure 9 shows respectively the absolute and relative execution times for operations performed by  $SC$  to obtain an encryption key. Figure 9 indicates that most of the execution time is spent on the decryption of the domain key and the call round trip time (call RTT). The absolute duration of the encryption key request is on average 0.025 seconds.

**Table 1: Execution times (in seconds) for some functions in the secure component, 1000 requests.**

Cumulative time	Per call	Function
3.300	0.001	RSA Encryption
10.445	0.010	Call RTT
9.001	0.009	RSA Decryption
24.974	0.025	Total Execution Time

In addition to that, we deployed an IaaS cluster using version “Havana” of OpenStack, a popular  $CP$  in order to measure the time needed for a VM to be launched. This time would then be compared with the time needed for the

generation of a domain key. As the process of key generation for the domain of a new VM instance is taking place in parallel with the VM launch, this comparison would be a good metric to see whether our protocol affects the performance of the *CP* or not. According to our measurements, the average time to launch a VM instance is 20.57 seconds while the average time for a domain key request is 0.025 seconds. Taking into consideration the fact that a domain key request will usually take place during the launch of a VM, our protocol does not affect the overall performance of the *CP*.

## 8. CONCLUSION

In this paper we have considered the problem of secure storage in IaaS environments. More precisely, we proposed a protocol that ensures confidentiality and integrity protection of stored information in a cloud environment. Furthermore, we presented an XML-based language framework that allows the clients of IaaS clouds to securely share their data and assign different access rights to users. The analysis was coupled with extensive experimental results which showed that the proposed language adds only a reasonable overhead to the operation of a cloud management platform. In our future work, we aim to improve the protocol and reduce the trust base by removing the need for a TTP. While this may affect the performance of the protocol, it would allow us to consider more complex attack scenarios which better reflect the complexity of information flow in IaaS clouds.

## 9. REFERENCES

- [1] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS '09*, pages 199–212, New York, NY, USA, 2009. ACM.
- [2] Juraj Somorovsky, Mario Heiderich, Meiko Jensen, Jörg Schwenk, Nils Gruschka, and Luigi Lo Iacono. All Your Clouds Are Belong to us: Security Analysis of Cloud Management Interfaces. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security, CCSW '11*, pages 3–14, New York, NY, USA, 2011. ACM.
- [3] Michael Jordon. Cleaning up dirty disks in the cloud. *Network Security*, 2012(10):12–15, 2012.
- [4] Dawn Song, Elaine Shi, Ian Fischer, and Umesh Shankar. Cloud data protection for the masses. *IEEE Computer*, 45(1):39–45, 2012.
- [5] M. Rezaei, NS Moosavi, H. Nemati, and R. Azmi. Tcvisor: A hypervisor level secure storage. In *Internet Technology and Secured Transactions (ICITST), 2010 International Conference for*, pages 1–9. IEEE, 2010.
- [6] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 203–216. ACM, 2011.
- [7] W. Wang, Z. Li, R. Owens, and B. Bhargava. Secure and efficient access to outsourced data. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 55–66. ACM, 2009.
- [8] S. Graf, P. Lang, S. Hohenadel, and M. Waldvogel. Versatile key management for secure cloud storage. *Submitted at EuroSys*, 11(11.4):2012–13, 2012.
- [9] M. Waldvogel, G. Caronni, D. Sun, N. Weiler, and B. Plattner. The versaKey framework: Versatile group key management. *Selected Areas in Communications, IEEE Journal on*, 17(9):1614–1631, 1999.
- [10] Nicolae Paladi, Christian Gehrman, and Fredric Morenius. Domain-Based Storage Protection (DBSP) in Public Infrastructure Clouds. In *Secure IT Systems*, pages 279–296. Springer, 2013.
- [11] Trusted Computing Group. TCG Specification, Architecture Overview, revision 1.4. Technical report, Trusted Computing Group, 2007.
- [12] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to Construct Random Functions. *J. ACM*, 33(4):792–807, August 1986.
- [13] Michael Hohmuth, Michael Peter, Hermann Härtig, and Jonathan S Shapiro. Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, page 22. ACM, 2004.
- [14] D. Dolev, Stanford University. Computer Science Dept, and A.C. Yao. *On the Security of Public Key Protocols*. Report (Stanford University. Computer Science Dept.). Department of Computer Science, Stanford University, 1981.
- [15] P. Mell and T. Grance. The NIST definition of cloud computing (draft). *NIST special publication*, 800, 2011.
- [16] Nicolae Paladi, Christian Gehrman, Mudassar Aslam, and Fredric Morenius. Trusted Launch of Virtual Machine Instances in Public IaaS Environments. In Taekyoung Kwon, Mun-Kyu Lee, and Daesung Kwon, editors, *Information Security and Cryptology – ICISC 2012*, volume 7839 of *Lecture Notes in Computer Science*, pages 309–323. Springer Berlin Heidelberg, 2013.